Optimization of slice encoding in Gobra

Bachelor's Thesis Project Description

Zdeněk Šnajdr
Supervised by Dionysios Spiliopoulos, Prof. Dr. Peter Müller
Department of Computer Science, ETH Zürich
Zürich, Switzerland

March 2023

1 Introduction

Gobra is an automated, modular verifier for Go code. It was developed by the Programming Methodology Group at ETH Zurich and serves as a frontend for Viper, translating annotated Go code into Viper AST and then utilizing the Viper verification language to perform formal verification. One of the key strengths of Gobra is its support for a large subset of the Go programming language, allowing developers to leverage the benefits of formal verification without sacrificing the flexibility and expressiveness of the Go language. By translating Go code into Viper, Gobra provides a rigorous way of verifying code correctness, helping to prevent errors, bugs, and other issues that can lead to software vulnerabilities. This makes it a valuable tool for developers who want to ensure the correctness, safety, and security of their code.

Viper Intermediate Language is designed to simplify the verification of programs that involve mutable state by providing built-in support for reasoning about program state using a form of separation logic. Viper has a syntax that is similar to C, and it supports both sequential and concurrent programming paradigms. Viper supports writing specifications and contracts. This makes it easy to verify programs using formal methods, and it enables developers to define and enforce correctness properties on their programs. The goal of Gobra is to improve the overall reliability and security of Go programs by reducing the risk of bugs or vulnerabilities.

2 Motivation

One of the data structures supported by Gobra are slices. A slice offers a dynamically-sized, flexible view of the elements of an array. Slices are a commonly used data structure in Go for a variety of tasks, such as manipulating and processing portions of data stored in large arrays.

In Go, there is no direct concept of a subarray like there is for a subslice. At the moment, subslices are part of the current enconding and the functionality for subslices is automatically added to the translated Viper code when there is syntax for slicing a slice present in the code. Nevertheless, there are numerous cases where no subslicing operation takes place. In such cases an alternative encoding of slices in Gobra might be a better fit. The goal of such approach is to use different encodings of slices depending on the requirements of each function. Specifically, we want to use a simpler encoding for functions that do not use any form of subslicing, and the current encoding with subslicing functionality for functions that have instances of subslicing. Optimizing the current encoding of slices in Gobra, with modified behavior for handling subslices, should improve the performance of programs that heavily use these data structures. The resulting time improvement in the verification process could impact the overall performance of the code.

Listing 1 serves as an illustrative example. In the last line of the main function there is a function call in the assignment with a subslice as a parameter. The function `sum` sums up the elements of the given slice. Notice that the generated Viper code includes the functionality for creating and handling subslices. In 2, the first axiom is triggered when we use `slen` and states that `o + c <= (ShArraylen(a):Int)` over the slice constructor. The second axiom states that `(soffset(s):Int) + (scap(s):Int) <= (ShArraylen ((sarray(s):ShArray[T])):Int)` for all `Slice[T]`. The offset is actually only needed for subslicing and

is often triggered and used even when it is unnecessary as both of these predicates can be dropped when no subslicing takes place in the code.

This additional functionality for subslicing represents an overhead during the verification process. However, there is no need to create a subslice for the function parameter since only the values pointed to by the slice are important. In fact, one could use an alternative encoding such that the given Gobra code translates into Viper without the need for subslicing. In this example, the idea is to verify the `sum` function with the simpler encoding and the `main` function with the current encoding.

Both encodings, the current one and the new one, are meant to be used in parallel. The first step towards implementing the new encoding would be to create a basic implementation in Gobra for all functions that do not use subslicing. This can be used to determine the performance of the new encoding against the existing encoding.

Once the new encoding has been tested, the next challenge would be to create a syntactic check that can determine which encoding is most appropriate for each function. This would involve analyzing the code to identify any instances of subslices and then deciding whether to use the current or new encoding based on the code requirements.

3 Core Goals

1. Design a simpler function-specific encoding for slices that does not support subslicing

   The current encoding represents verification overhead when subslicing is not needed. The simpler encoding solves this issue by leaving out the support for subslicing for functions that do not need it.

2. Prototype the said encoding in Gobra

   This will replace the current encoding just for evaluating the performance of the new encoding in functions where it is suitable.

3. Evaluate the speedup compared to the current encoding

4. Implement syntactic check for choosing the correct encoding for the method

   The purpose of the syntactic check is to decide if the simpler encoding can be used for a particular function. Otherwise the current encoding will be used.

4 Extension Goals

1. Investigate if the slice encoding can be chosen in a more fine-grained manner instead of per function

2. Explore more function-specific encodings

References

[1] *A Tour of Go*. `https://go.dev/tour/list`.

[2] *Gobra*. `https://github.com/viperproject/gobra`.

[3] *Viper Tutorial*. `https://viper.ethz.ch/tutorial/`.

Listing 1: Gobra

```
package main

func main() {
    nums@ := [5]int{1, 2, 3, 4, 5}
    slice := nums[0:]
    s := sum(slice[1:4])
}

requires forall k int :: 0 <= k && k < len(x) ==> acc(&x[k])
ensures forall k int :: 0 <= k && k < len(x) ==> acc(&x[k])
func sum(x []int) (res int) {
    var length = len(x)
    sum := 0

    invariant 0 <= i && i <= len(x)
    invariant forall k int :: 0 <= k && k < len(x) ==> acc(&x[k])
    for i := 0; i < length; i++ {
        sum += x[i]
    }
    return sum
}
```

Listing 2: Excerpt from the Slice domain

```
axiom deconstructor_over_constructor_len {
    (forall a: ShArray[T], o: Int, l: Int, c: Int ::
      { (slen((smake(a, o, l, c): Slice[T])): Int) }
      0 <= o && (0 <= l && (l <= c && o + c <= (ShArraylen(a): Int))) ==>
      (slen((smake(a, o, l, c): Slice[T])): Int) == l)
  }

axiom {
    (forall s: Slice[T] ::
      { (soffset(s): Int), (scap(s): Int) }
      { (ShArraylen((sarray(s): ShArray[T])): Int) }
      (soffset(s): Int) + (scap(s): Int) <=
      (ShArraylen((sarray(s): ShArray[T])): Int))
  }
```